CHAPTER 16

Building Asynchronous Message-Oriented Applications with Microsoft Message Queue Server

In This Chapter

- Working with messages
- Understanding MSMQ as a whole
- Defining MSMQ business requirements
- Insights into the limits MSMQ imposes
- Working with the MSMQ API
- Working with transactions
- Dealing with application errors

We took a preliminary look at Microsoft Message Queue (MSMQ) in the previous chapter. It's important to know that you normally use MSMQ in combination with Microsoft Transaction Server (MTS) for certain classes of applications, especially those that require high reliability like database management systems (DBMS). However, it's just as important to know that MSMQ is a standalone product that you can use for nontransaction-oriented applications that simply require the capability to work with messages.

E-mail is a perfect example of a nontransaction-oriented application that you often need to work with in a disconnected state. Getting e-mail ready offline, then getting online just long enough to transmit the data to a server is one of the common tasks that we've all performed. Obviously, there are enough e-mail packages on the market that you don't need MSMQ to create a custom e-mail package for your company. In most cases, text transfer between clients is better accomplished with one of these prepackaged applications.

However, there are types of e-mail-like applications that you may want to create using MSMQ. For example, network administrators may want to track the status of all of the machines under their control. This is a relatively easy task to perform on the local network. All you need to do is load an agent on the client machine, then use some type of server-based product to query the agents for status information. Monitoring packages today can tell an administrator everything from what applications a user currently is working with to the speed of the cooling fan.

Laptops form a special class of machine because they're used extensively on the road where the network administrator can't get in contact with them. How can a network administrator keep track of these machines while on the road? A message transmitted each time the user makes contact with the home network is one answer. Just like the agent on desktop machines, an agent on the laptop can make log entries that get transferred to a queue. When the user is connected to the home network, these log entries automatically appear on the server immediately after they're made. On the other hand, when the user is on the road, the log entries appear on the server when the user makes contact. As far as the user and the agent are concerned, everything works the same no matter where the user is located physically. MSMQ provides the "magic" that allows a network administrator to track machine conditions even when that machine is on the road.

Obviously, before you can implement a solution as I just described, you need to know what messages are all about. The first section of this chapter provides you with a very quick understand of what messages are and why they're useful. The messages that MSMQ provides contain much more than just the data you need to move from one place to another.

Once you understand the basic unit of data transfer in MSMQ, you'll want to know something about how MSMQ works at a low level. The second section of this chapter provides the detailed theory you need in order to use MSMQ fully to create all kinds of message-oriented applications. The ability to create disconnected applications that rely on messages rather than a direct connection opens a wealth of new situations where you can use this technology. Just as agents normally maintain direct contact with the server, yet require a message orientation when used in disconnected mode, there are other applications where you normally may use a direct connection but that can be extended using messages.

Building a business case for MSMQ is also important because this is a new technology. Many companies implement a new business strategy based on what that strategy can do for them. The third section of this chapter helps you build this

business strategy, making it easier for management to see the need for using MSMQ in the applications you create.

There aren't any perfect technologies. Just like any other solution to programming problems, MSMQ comes with its own set of limitations. In the next section of this chapter, I'll discuss those limitations and some of the ways you can work around them.

MSMQ, like many Microsoft technologies, provides more than one way to get the job done. The fifth section of this chapter looks at the MSMQ API. You can access this API using the newer COM component method, or the more traditional API function call approach. Obviously, the function call approach is useful only if you're a C++ programmer (or a very able Visual Basic programmer).

As previously stated, MSMQ allows you to work in standalone mode or with MTS. What you may not know is that there's more than one way to create a transaction with MSMQ, some of which don't require MTS. In fact, there are quite a few transactional data-transfer options that you need to consider. The sixth section of this chapter helps you understand MSMQ transactions and the various options you have when using them.

Even the most reliable technology in the world experiences problems, some of which are unavoidable because the user causes them. Knowing how to handle exceptions is an important part of MSMQ programming. In the last section of this chapter, I'll talk about how to handle exceptions and some of the ways that you can avoid them in the first place.

A lot of the latest MSMQ features require you to install the most recent version of the Windows Platform SDK. In fact, many of these features are totally undocumented outside of the Windows Platform SDK. What this means is that Microsoft's latest Visual Studio release doesn't handle MSMQ programming tasks very well without this additional SDK support. I assume that you've Visual Studio Service Pack 3 and the latest version of the Windows Platform SDK installed as a minimum when working with the coding examples in this chapter.

You can download Visual Studio Service Pack 3 (or later if available) from the Visual Studio Web site at `http://msdn.microsoft.com/vstudio/sp/default.asp`. The latest version of the Platform SDK appears at `http://msdn.microsoft.com/developer/sdk/platform.asp`. I use Visual Studio 6.0 Enterprise Edition for the examples in this chapter; other versions aren't guaranteed to work.

Understanding Messages

In Chapter 14, I spent a lot of time talking about DCOM and how it works. In some ways, MSMQ works similar to DCOM. What you're doing is creating a connection between a client and server for the purpose of information exchange. However,

instead of creating a direct connection using a network protocol like DCOM does, MSMQ uses queues as an intermediary destination for the data that flows between the client and server. These queues are used to hold messages, which is the method that MSMQ uses in place of the packets normally used by networks to transfer data.

The advantage to using the message as a basis for transferring information is that the client and server don't need to exist at the same time. In other words, users can open their applications without checking for server availability first and they won't even notice if the server is available once the application is open. A salesperson can record an order now, then allow MSMQ to upload it automatically to the server later without modifying the methods that he or she normally would use. MSMQ makes the delivery in the background without any interaction on the part of the user. The orders that the salesperson creates while the connection to the server is severed are stored in a local queue as messages. Each message normally represents a single order's data, though there really isn't a limit on how you can format the messages. A message can hold more than one message or only part of a large order.

The disadvantage of using messages as the basis for data transfer is that there's some additional overhead. Obviously, there are more layers of processing now, which means that direct connect scenarios actually run a bit slower. In addition, there's a little additional coding and setup time for the developer when using MSMQ. However, the advantages far outweigh the disadvantages — you'll see why as the chapter progresses.

So, what does a message contain? All messages are objects that contain the same number of properties, though you won't necessarily use all of the properties all of the time. Table 16-1 contains a description of the various message properties that you'll use most often.

Table 16-1
MSMQ Message Properties

Property	*Description*
Additional Queue Information	
Ack	Defines the kind of message that MSMQ places in the Acknowledgement queue. The default setting doesn't place any messages in the Acknowledgement queue at all. You can also choose options that place messages in the Acknowledgement queue based on when the message is received by the queue or retrieved by the listener.
AdminQueueInfo	Specifies the name and location of the Acknowledgement queue.
DestinationQueueInfo	Tells MSMQ where you want the message sent. This is a read-only property that's set by MSMQ after you specify a destination using a method argument.

Property	*Description*
Journal	Determines whether MSMQ stores a copy of the message in the system journal.
ResponseQueueInfo	You can set MSMQ to provide a response when it receives your message. This property determines the location of the Response queue on the client machine. I talk about this property more in the "Understand MSMQ Limitations" section of the chapter.
Trace	Determines if the message is traced as it travels from machine to machine. The tracing messages are placed in the Report queue.
Data	
Body	Contains the message that you want to transfer from one machine to another. This message is stored as a byte array, which means that you can transfer a wide variety of data types, including objects and strings. You definitely want to consider the method used to pack data because you need to unpack it later.
BodyLength	Defines the size of the message body.
Miscellaneous	
AppSpecific	Allows you to add application-specific information to the message.
Class	This is a read-only property set by MSMQ that specifies the message type. A message can be a standard MSMQ message, a positive or negative acknowledgement, or a report.
CorrelationId	An application-generated number that relates this message to all other messages sent during a given timeframe. It allows the application to sort the messages when they arrive at the receiving queue.
Id	A read-only property that contains the MSMQ-assigned identification number for the message.
Label	An application-specific moniker for the message. This property can be used for a variety of purposes, but is normally used for displaying a message header in human-readable format.

Continued

Table 16-1 (*continued*)

Property	***Description***
Priority	The priority determines the importance of the message when compared to other messages. MSMQ uses values from 0 to 7, with 0 being the highest. The default priority setting is 3. Transactional messages are automatically given a priority of 0 by MSMQ.
Security	
AuthLevel	Tells MSMQ whether you want the message authenticated when it arrives at the queue.
EncryptAlgorithm	Defines the algorithm used to encrypt the message prior to transmission. A higher level of encryption provides greater security, but also increases the size of the resulting message.
HashAlgorithm	Determines the hash algorithm used to authenticate a message.
IsAuthenticated	A read-only property that tells whether MSMQ authenticated the message.
PrivLevel	This property tells MSMQ whether the message is private or public. There are three levels of privacy available in the United States: 40-bit, 40-bit enhanced, and 128-bit message body encryption. (The levels of privacy available to you are affected by your current location.) The last two levels of encryption are available only when using MSMQ 2.0, which means that you can use them only with Windows 2000.
SenderCertificate	Represents the message sender's security certificate within a byte array. The security certificate is used to authenticate the message.
SenderId	Provides the identity of the person who sent the message. This read-only property is set by MSMQ when the message is sent, unless you specifically declare that the message won't be authenticated.
SenderIdType	Defines the type of sender identification: none or a standard sender ID (SID).
SourceMachineGuid	Contains the globally unique identifier (GUID) for the sending machine. This is a read-only value that's automatically added by MSMQ.

Property	*Description*
Time Tracking	
ArrivedTime	A read-only property that specifies when the message arrives at the queue.
MaxTimeToReachQueue	Defines the amount of time that the message has to reach the queue.
MaxTimeToReceive	Defines the amount of time that the listener has to remove the message from the queue for processing.
SentTime	Contains the time that the message was sent by the client. This is a read-only value that MSMQ automatically adds.
Transactions	
Delivery	Defines how you want the message delivered. See the "Understanding Transactions" section for more details.

An Overview of MSMQ

So, how does a messaging application differ from DCOM? The data most likely is the same, which means that your business logic is the same. However, the delivery method is different. We're now using the idea of messages to transfer data and a queue to hold those messages (a queue is a sort of a mailbox for messages). As a minimum, there's a queue on the server that holds all of the messages that the server components process. Each active component on the server has a separate queue. These components pick messages up from their queue whenever processing on the current message is complete.

In addition to the server queue, some clients have a queue as well for local processing. The local queue gets emptied into the server queue anytime there's a connection between the client and the server. In short, messages can be viewed as the packets that normally get carried between client and server on a network. Obviously, this is a simplification of a more complex process, but it does help to start out with this perspective of the functioning of MSMQ and the messages that it uses.

The obvious advantage for using MSMQ in place of DCOM is the ability to perform disconnected application handling. The local client queue allows the client to continue processing information even when there's no direct connection to the server for handling the messages. (Obviously, you need to configure the client for independent use.) Using a local client queue allows the user to continue working as if the connection existed and without performing any special procedures to transfer the resulting messages from the client to the server when a connection is made. MSMQ handles all of the message transfers in the background without the user's

knowledge. Figure 16-1 shows the relationship between the client and the server, and the kinds of queue setups that you can expect.

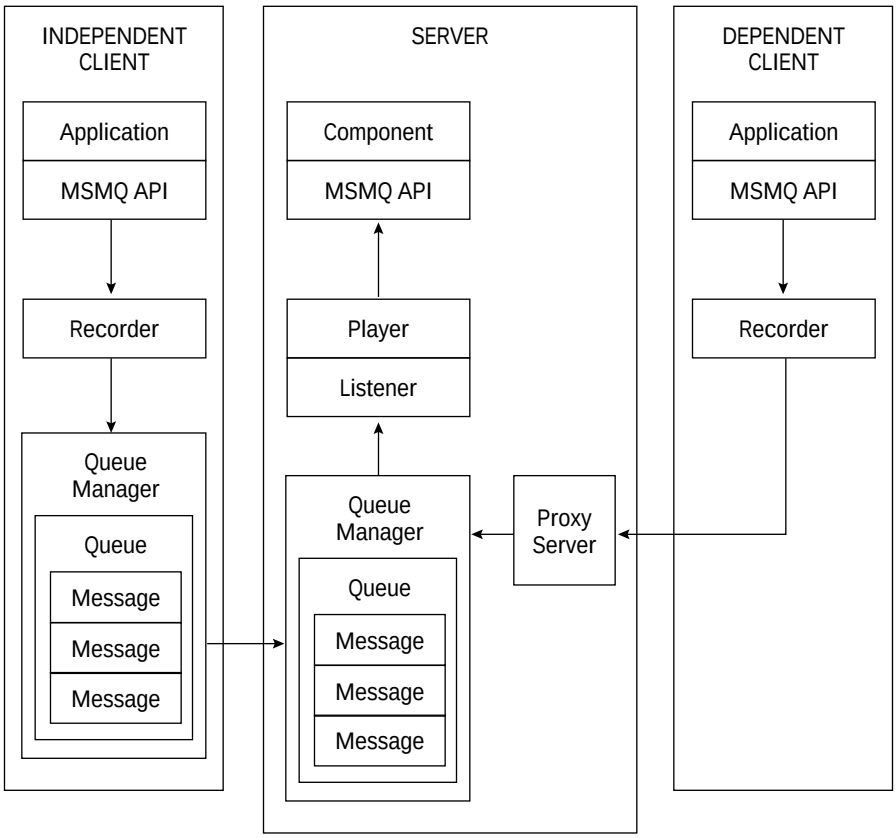

Figure 16-1: MSMQ allows the user to continue working even when no connection between the client and server exists.

Notice that there are actually two client types: dependent and independent. The main difference between the two is that the independent client also provides its own queue, whereas the dependent client relies on a direct connection to the server. Obviously, you can't use a dependent client setup for a laptop computer that you intend to use on the road because the dependent client lacks a queue. However, a dependent client setup works for a desktop machine that's located in the same building as a server or on a WAN with a reliable connection. Using a dependent client setup reduces the disk requirements for using MSMQ and can result in a slight performance boost because the messages are only placed in one queue rather than two.

There are three elements required for establishing and maintaining a message flow between the client and server: recorder, listener, and player. The *recorder* takes the client output, creates messages, then places those messages in either the local message or sends them to a proxy server on the server. MSMQ takes the message that the client creates and places it in the server's queue. When the *listener* sees a message in the server's queue, it removes the message and gives it to the player. Finally, the *player* takes the message and turns it into data for the server.

You can look at this process as the same one that goes on with an answering machine in your home. When someone calls and finds that you're not home, they leave a message by talking to the answering machine instead of you directly. The answering machine stores the message using any number of methods. When you get home, an indicator on the answering machine tells you that you have one or more messages. Pressing a button on the answering machine normally plays the messages back for you, allowing to you to determine who has called in your absence. As you can see, the idea of disconnected communication isn't new; MSMQ represents a new implementation of an existing idea.

In Chapter 15, I talk extensively about transactions. It turns out that MSMQ applications use a minimum of three transactions for every data transmission, even if it appears that there's only one transaction taking place. The first transaction occurs between the client application and the local queue. MSMQ creates a second transaction when it takes the message from the local queue and places it in the server queue. That's where the third transaction begins. As soon as the server removes a message from the queue, it creates a third transaction that tracks the message's progress on the server. This three-transaction approach makes it a lot less likely that an update will fail due to communication problems, which, in turn, makes the application more reliable. In addition, since there's a special MSMQ transaction for delivering data from the client to the server, you can be sure that each message gets transmitted successfully only one time, but that it does get transmitted at least one time.

Throughout this chapter, I'll address the topic of queues in several ways. There are two kinds of queues supported by MSMQ: application and system. The application queue is created by the application and is used for messages, administration, reports, and responses. Systems queues are created by MSMQ. There are two types: dead letter and journal. Queues can also be public or private. Public queues are available for anyone to use and are tracked by the *MSMQ Information Service (MQIS)*. Private queues are normally used for one-to-one communications such as a response from a server on a client machine. I'll talk more about these queues as the chapter progresses.

The Business Case for MSMQ

Now that you understand what's going on with MSMQ a little better, it's important to understand how MSQM can help you in a practical business way. Sure, being able to complete transactions while the client is disconnected from the server is an

important way to make applications reliable, but what are the hidden benefits that you can use to sell your boss on the idea? The following sections provide an overview of some of the business reasons to use MSMQ in place of some of the other technologies that you may have tried in the past.

User Training

Right now, you have to train users to perform their tasks in at least two different ways if you want to implement a disconnected application scenario for mobile computers. They have to use one technique when they're directly connected to the server and a second (two-step) technique when disconnected. Of course, this means that you have to invest more time and money training these users, which increases business costs. In addition, since there are two techniques for performing a single task, the probability of user confusion and the resulting user error is much higher.

Reliable Database Updates

Mobile users are currently required to perform transactions in two phases. First, they take the order at the remote location from a new customer. Second, they have to update all of their orders to the company database in some way. This two-step approach to working with transactions means that the potential for human error greatly increases.

Several problems can occur, including the loss of data between the time the order is taken and the time it arrives at the company server, the user simply forgets to upload his or her orders, the order is mangled during transmission, or the order gets transmitted more than once. Because of the way MSMQ is designed, the update process not only is automatic, but the use of transactions ensures data integrity. In short, you spend a lot less time troubleshooting errant database entries.

Application Scaling

One of the MSMQ features that you may not notice immediately is that the disconnected processing scenario also allows the server to scale better. When using a direct connection, the client requests have to be processed immediately or the client will notice a processing delay. Using MSMQ doesn't mean that the requests get processed any faster, but placing the requests in a queue allows the call to return faster, which means that users won't notice a processing delay nearly as fast. Users can continue to work as if they have the server all to themselves. When the processing load starts to fall, the server can begin to process the messages contained in the queues at a faster rate. As a result, the server runs at a higher efficiency and the users are happier.

Noncritical Path Processing

You can also use MSMQ to prioritize the work that the server needs to perform. For example, the server has to be available to receive customer orders all of the time. Otherwise, you may lose an order to a competitor who is ready. However, once you have the order in hand, you don't need to process it immediately. You can check the customer's credit card validity during off peak times. In addition, order fulfillment can take place during the evening hours when the server is less likely to be busy. All this means that you get better efficiency out of your server investment by ensuring that the server is always busy doing some type of work for your company.

Fewer Error Messages

Consider for a moment, the cost of an error message to your business. The user has to call a support technician. If the support technician isn't available immediately, then the user sits waiting instead of processing orders for your business. The more error messages that a server generates, the greater the number of support technicians you require and the more time the user wastes waiting for an answer. Because MSMQ allows your server to scale better, the user receives fewer resource-related error messages, meaning that the ability to scale affects your bottom line in several easy-to-understand ways.

Human Processing Time Removed

When you create a direct connection between the client and the server, the objects created on the server usually exist for the entire time the application runs on the client machine. This means that the server is allocating resources while the user is thinking about the data needed to commit a transaction. Using MSMQ means that the human part of the equation is removed from server resource allocation. The server-side objects are created only long enough to process the transaction after all of the data is collected. As a result, the server uses resources more efficiently and usually can handle a larger processing load.

Understanding MSMQ Limitations

MSMQ doesn't fix every programming problem you may have with your application today and, because of the way it's designed, it actually introduces a few new limitations that you need to know. In short, MSMQ is another tool for your toolbox, but it shouldn't be viewed as the only tool. What Microsoft is attempting to do is create an environment where more than one tool may meet a particular need; you need to consider the pros and cons of each tool before you use it. The following list helps you understand MSMQ limitations.

One-Way Message Transport	Previous chapters looked at components that require one or more inputs and produce some type of output. However, when you work with MSMQ, you can create components only with inputs and no outputs. The reason for this one-way transfer of information is easy to understand — the client and server may not be available at the same time. As a result, the server may not have a client to which it can send a response.
HRESULT Value Limited	Because the client may send a message to the server in disconnected mode, you can't expect the same kind of HRESULT that you've gotten in the past. When using MSMQ, an S_OK return may indicate that the data was received by the local queue instead of telling you that the transaction succeeded as a whole. This means that you have to limit your assumptions when designing the application.
No Synchronous Result	MSMQ provides asynchronous data transfer through messages. This means you can't expect instantaneous results from it. For example, the user may query the database for recent customer orders. If you store that information on the user's local hard drive, then you can expect to get a response even in disconnected mode. On the other hand, if the information appears only on the server, then you need to provide users with some type of informational feedback that allows them to retrieve the information they need. In short, there are some situations when a connection is required. The best you can do is to download a subset of the company's database to the user's machine — those records that you think the user is most likely to need while in disconnected mode. This means that MSMQ applications require some level of intelligent download capability based on the user's schedule.

Complete Messages Only	Since the requestor may be unavailable when the server begins processing a request, the messages you send to the server's message queue have to contain complete information. For example, if you want to enter a new order into the fulfillment database, then all of the data required for that order have to appear within a single message. Unfortunately (despite assurances to the contrary), this may require some recoding on your part to get an application ready for MSMQ use.
Data by Value, Not by Reference	C programmers as a whole are accustomed to using pointers for everything. That doesn't work with MSMQ. You must send data by value, not by reference. The reason is fairly obvious when you think about it. A lack of direct connectivity means that the reference is worthless. Only discrete values can provide the information that the server requires.

As you can see, most of the limitations of MSMQ are based on the same thing that makes this technology so attractive in the first place. When you create a disconnected application, you have to assume certain things about the client and the server, like the inability of the server and client to talk with each other. This means that you have to think about the kinds of information the client and server require well in advance of the first trial on the road.

You can partially overcome the problems of disconnected applications by creating two different component message queues: one on the client and another on the server. Since message flow is a one-way process, you need two separate sets of components (or dual-purpose components) to implement this idea. Figure 16-2 shows one way that you can use this methodology to help support a user's need for customer information on the road.

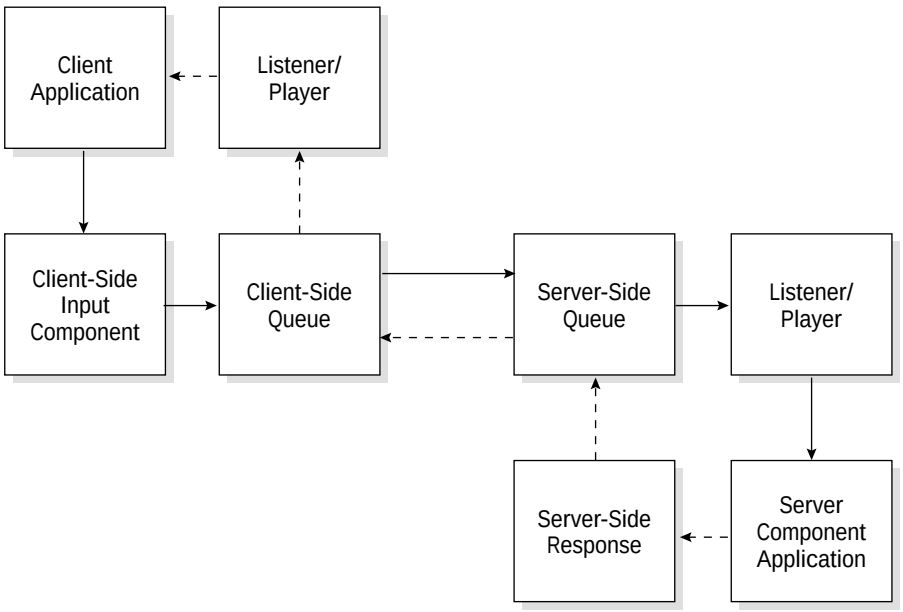

Figure 16-2: MSMQ does provide ways of creating interactive messaging; you just have to be creative in your approach.

As shown in Figure 16-2, this method does require an independent client with its own queue. In addition, the client and the server require a component to manage input to the queue. The client can have a listener component that allows the server to send it messages that respond to client requests. Likewise, the server needs a listener component to respond to client input. (The client-to-server message route appears as solid lines, whereas the server-to-client route appears in dashed lines.)

Obviously, you still don't get anything close to real-time response using this method, but it does allow for smarter download of user-requested data. For example, a user can place a list of names for the next day's sales meetings in a client-side queue. When a connection to the server gets created, the client-side queue gets transferred to the server. The server, in turn, can create a list of local client database entries based on the meeting list and download it to its local queue. The listener on the client receives these new database entries and updates the client's local database in the background as the user completes other tasks. In short, the user always has access to the next day's client records from the company database even though those records normally require a direct connection. This is an example of smart application downloads. The one-way message connection can be overcome by using multiple components designed to provide a variety of services on both the client and server.

So, at this point, you're probably thinking that you may not be able to use MSMQ because of your application's need to receive a response for every transaction. Actually, there are a lot of situations when the client machine doesn't require a response. For example, when salespeople input orders, they really don't need any response from the server. In most cases, a salesperson simply assumes that the order made it to the server. If an update on order status is required, a salesperson can always issue a status request that gets answered during a later connection. As an alternative, the server can always assume that it needs to download active order status information and provide it during the user's regular connection time. In short, there are a lot of times when you simply don't need an instantaneous response to server input — an update the next day works just as well.

There are going to be times when you absolutely have to have some kind of response to a request. For example, what if an exception occurs during order processing? The salesperson may have forgotten to include the customer ID on an order. In this case, there's another method that MSMQ provides for dealing with the situation. You can pass a reference to a response object on the client machine as part of your request. The messages are still passed one way, and you still need two separate components to achieve two-way data transfer. In this case, however, the response is sent automatically. Obviously, you want to limit this kind of processing so that the connection between the client and server isn't overwhelmed during intermittent user contacts.

Understanding the API

Windows 2000 actually provides two methods for working with MSMQ. You get to choose between the API access method used by previous versions of MSMQ or the newer COM component route. Either route allows you to create, delete, and work with queues. In most cases, the decision of using either API calls or COM components is a matter of personal taste. However, the big advantage to using COM components is that you can use them with a variety of languages. For the most part, you're limited to using Visual C++ when working with the MSMQ API. On the other hand, some programmers feel more comfortable using the MSMQ API because they're more familiar with standard C++ syntax. Since the COM components are newer, somewhat easier to work with, and much more flexible than the API calls, let's concentrate on using the COM components in this chapter. Table 16-2 talks about the various COM components and how you use them in an application.

Table 16-2
MSMQ COM Component Summary

Component	*Description*
MSMQApplication	Allows you to perform two machine-related tasks. First, you can obtain the ID of the machine using the MachineIdOfMachineName( ) method. Second, you can register a certification that's used for security purposes using the RegisterCertification( ) method.
MSMQCoordinatedTransactionDispenser	Creates an external transaction that relies on MTS to perform the work. The output from the one method associated with this class, BeginTransaction( ), is a MSMQTransaction object, which you can then use for sending and receiving messages within a transaction. I'll talk more about the dynamics of transactions in the "Understanding Transactions" section that follows.
MSMQEvent	Provides an event-handling capability that allows you to monitor various MSMQ events, then react to them within a listener (or other) component. This component allows you to monitor messages arriving at the queue, errors that occur while a message is being delivered to the queue, and messages that don't arrive at the queue before their time-out timer expires. Using the MSMQEvent component allows you to write a single component that can be associated with any number of queues to handle common message-related tasks.
MSMQMessage	Contains all of the methods and properties normally required to work with new messages. You use this component to send messages, as well as gain access to a security context for the message. The properties associated with this component allow you to define all message characteristics completely, including the administration and response queue locations. You can also define encryption and delivery options. We'll look at more message-related features as the chapter progresses.

Component	*Description*
MSMQQuery	Use this component to ask Active Directory about public queues. The single associated method, LookupQueue( ), accepts a data structure that allows you to define the parameters of your search. The results of the search are returned in an MSMQQueueInfos object.
MSMQQueue	Allows you to work with queues. This component contains methods for receiving messages, peeking at the contents of the queue, closing the queue, and resetting the queue. A special method, EnableNotification( ), allows you to set up event-driven message processing for the queue.
MSMQQueueInfo	Use this component to perform queue management tasks. You can create a new queue, open or delete an existing queue, or refresh or update queue properties for the queue. This component also provides access to a wealth of queue-related properties such as privacy and priority levels, creation time, transactional state, time of the last modification, and maximum queue size.
MSMQQueueInfos	An object returned by the MSMQQuery object that allows you to search through a list of public queues meeting certain search criteria. This component relies on two methods, Next( ) and Reset( ), to manage the position of a cursor within the list of queues. The current queue is returned as an MSMQQueueInfo component that contains properties identifying the selected public queue.
MSMQTransaction	An object returned by a call to the BeginTransaction( ) method of either the MSMQTransactionDispenser or MSMQCoordinatedTransactionDispenser components. The three methods of this component allow you to commit, abort, or initiate a new transaction. You normally use this object as part of a MSMQMessage.Send( ), MSMQQueue.Receive( ), or MSMQReceiveCurrent( ) call.
MSMQTransactionDispenser	Creates an internal transaction that relies on MSMQ to perform the work. The output from the one method associated with this class, BeginTransaction( ), is a MSMQTransaction object, which you then can use for sending and receiving messages within a transaction. I'll talk more about the dynamics of transactions in the "Understanding Transactions" section that follows.

Understanding Transactions

Let's talk about transactions in more detail. There are two ways to deliver your message: fast or reliable. The fast method sends the message from machine to machine using memory alone, whereas the reliable method flushes each message to disk before sending it on to the next machine. The main difference between fast and reliable transactions is disk access. The disk access time may make the message delivery slower, but it also means that you can retrieve another copy of the message should it get lost between two computers or if a server failure occurs. MSMQ assumes that you want fast over reliable, so it normally uses the MQMSG_DELIVERY_EXPRESS option for your messages. If you want to use the slower, more reliable method, use the MQMSG_DELIVERY_RELIABLE option instead.

Whether you use the fast or reliable message delivery methods, the kind of queue you use is also important. Transactions require transactional queues. You can't send a transactional message to a normal queue and vice versa. MSMQ always requires you to determine what kind of queue you want to create during the creation process as shown in Figure 16-3. All you need to do is check the Transactional option in the Queue Name dialog box to make the queue a transactional queue. Obviously, this means that you have to decide whether you want to use transactions during the initial setup of your project or face the possibility of recreating the queues your application requires later.

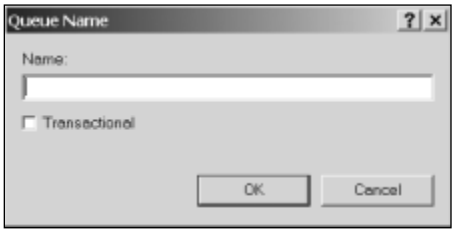

Figure 16-3: MSMQ requires that you determine what kind of queue you need during the creation process—you can't change the settings later.

MSMQ actually supports two transaction types: internal and external. The internal transaction relies on features provided by the MSMQ transport to ensure delivery, whereas the external transaction relies on MTS. The issue of which transaction type to choose is one of speed versus reliability. An internal transaction runs faster, but provides fewer features than an external transaction.

There are also two kinds of internal transaction types to choose from. You can use the MQ_SINGLE_MESSAGE option to send a single message using a transaction with the MSMQMessage.Send( ) method. If you need to send more than one message using a transaction, then you want to begin by creating a MSMQTransaction object.

Besides speed and reliability, there are other differences between internal and external transactions. One of the more important differences from a security standpoint is isolation. MTS transactions run at a serializable isolation level. This means that no one else can see the transaction—it's totally hidden from anyone but the client and server. A MSMQ transaction runs at the read committed isolation level. In other words, there are certain circumstances when a listener or other component can see more than just the current transaction. In short, it's a small breach of security, but still a significant one.

Another potential problem is that other resource managers can't use internal MSMQ transactions. For example, if you want to create a transaction that encompasses both MSMQ and SQL Server, you need to use an external transaction because SQL Server can't participate in an internal transaction. Of course, the lack of resource management coordination also means that internal transactions use less memory than a similar external transaction. Even a small savings can mean a significant difference when your server is processing hundreds (or even thousands) of transactions every hour. In short, the choice is one of resource management versus transaction flexibility in this case. The flexibility that MTS provides comes at the price of system memory and performance.

Handling Exceptions

Like anything else, there are situations when the default error handling isn't going to take care of your message queue. For example, a user may inadvertently generate a message with incomplete information, and no amount of rework can fix the message so that it can be processed later. Since the message is handled as part of a transaction and there's no client to consult with to determine how to handle the errant message, an aborted message simply ends up at the top of the message queue again. In short, your application can end up in an endless loop trying to process a message that can't be processed.

MSMQ provides a mechanism for dealing with messages that abort in the form of retry queues. What you're really doing is moving the message from the active queue to a background queue for processing later. This means that current messages don't get held up waiting for the errant message to process. Of course, you eventually need to handle the messages in the retry queue. At this point, there are three ways to handle errant messages as listed here.

Fix the Message	Sometimes the message lacks an important bit of information that you can add to make it process correctly. All you need to do, in this case, is fix the erroneous information and move the message back to the normal processing queue from the retry queue. In cases where there's a configuration issue (for example, the user lacks sufficient rights), an administrator manually can fix the problem and then move the message from the retry queue to the normal processing queue.
Gather Diagnostic Information and Delete	There are situations when no matter what you do to the message it doesn't work. In this case, you need to gather as much information about the message as possible, then delete it to prevent the message from becoming a problem for the rest of the system. Obviously, you want to generate an event log report using the diagnostic information you've gathered. This will alert the network administrator to potential problems with the MSMQ setup. In addition, you might need to generate an error message for the user so that he or she knows that the message failed. In some cases, you may be able to regenerate the message by having the user reenter the information. Finally, you might need to alert the component developer (or an in-house developer) so that the component can be updated and possibly repaired.
Generate a Compensating Message	This is a situation where the message can't complete because of outside influences. For example, you may have an order entry system that relies on a customer entering a credit card number for payment purposes. The card may be full or out of date, or perhaps the customer has entered the name incorrectly. A compensating message allows you to send a request for additional information to the customer, fix the message, then place it back into the normal processing queue. As an alternative, you can generate an error message and simply back the transaction out, effectively deleting the entire transaction from the system.

Implementing this error-handling capability requires a number of component configuration changes. The most important configuration item is specifying a queuing exception class for the component. This class has to implement the IPlaybackControl interface, which contains two methods, FinalClientRetry( ) and FinalServerRetry( ), along with any interface methods that the component normally provides. This exception-handing component gives you the opportunity to take care of message problems. If you decide that the message can't be fixed, then call FinalClientRetry( ) or FinalServerRetry( ) to place the message in the dead message queue (just like the dead letter office for the post office — messages enter but don't leave in most cases).

Summary

This chapter looks at the problems of creating reliable and easy-to-use applications in a world that's more mobile than ever before. Disconnected application capability is no longer a luxury that companies can afford to ignore. If you want to remain competitive, you have to provide some method of keeping the user on the road in constant contact with the company. In short, MSMQ provides the means to keep the mobile and home user connected and the company's database up to date. Here are some additional conclusions you can draw from the contents of this chapter.

- The easiest way to look at MSMQ is as a form of DCOM that uses queues as an intermediary destination for data flowing between the client and server.
- MSMQ provides some important business benefits that can affect your company's bottom line, including reduced user training costs, improved data integrity, enhanced server scaling, noncritical path processing, and reduced support costs for resource-related, server-side error messages.
- There are some limits to using MSMQ, most of which have to do with the disconnected nature of this technology. So, though a disconnected application does allow employees on the road to keep working, the same technology poses limits for the developer that may require creative solutions.
- MQIS is the central database of MSMQ information. All public queues are tracked within this database, making it possible to retrieve any required information about public queues from within your application.
- MSMQ is a valuable part of any business strategy because it reduces user training costs, provides reliable database updates, enhances application scaling, prioritizes the server's workload, reduces error messages, and removes human processing time from the picture.
- There are certain limitations that you may experience when using MSMQ. The most important limitations include one-way message transport, limited return value, and no synchronous result. In addition, you're limited to complete messages, and the data has to be passed by value instead of by reference. Most of these problems are incurred by the very feature that makes MSMQ so beneficial — disconnected processing.

- MSMQ provides two methods of access by outside applications: API and COM component. Of the two, the COM component is more flexible, easier to understand, and usable by more languages.
- The two common methods that a listener uses to detect new messages in a queue is event processing and polling. The polling method is self-contained and requires no additional coding. The event-processing method requires you to create a separate event handler (MSMQEvent) component.
- There are two methods of transferring messages in MSMQ: express or reliable. An express message gets transmitted using memory only, which makes it move very fast from computer to computer. On the other hand, a reliable delivery is flushed to disk at every computer — making the transfer slower, but easier to recover.
- MSMQ also supports internal and external transactions. An internal transaction relies on the MSMQ transport. This means it incurs less overhead, but places limitations on how the transaction can be used. On the other hand, an external transaction relies on MTS — so it's more reliable, but also slower.
- Exception handling is an important part of every programming effort. When using MSMQ, you can handle exceptions in one of three ways: fix the message, gather diagnostic information and delete the message, or generate a compensating message.

✦ ✦ ✦